Alex Chen

CS 31

Project 4 Write up

a.

This project held the most obstacles for myself out of the four projects I have accomplished. The largest challenge in this project was dealing with all types of passed arguments. Having a Java background, I have been accustomed to simpler methods and concerns during looping through arrays. There typically is not a number of elements that is passed to allow one to loop through an array. This opens up the possibilities of n not being equal to the length of the array, which means there are many other cases to consider. One such occurrence was during writing the subsequence method. During this I initially only failed the program if n1 or n2 was negative. However, I realized there are immediate values that could be returned for other cases of n1 and n2 values. For one, n2 cannot be larger than n1, or else no subsequence can exist. Furthermore, if n2 is equal to 0, then the subsequence begins at index 0, based on logic. Lastly, if n2 is not equal to 0, but n1 is equal to 0, then the n2 is definitely not a subsequence of n1 and -1 is returned. Such ideas need to be considered in the world of C++, which definitely made me spend a lot more time double and triple checking the logic in each function.

One function I distinctly remember having trouble with is separate. The aspect of writing separate that was troubling past logic was, like subsequence, writing a function that could properly evaluate all the possible cases past the basic cases. Specifically, I was able to write separate to deal with all cases except for when a string equal to separator was actually present in the array. Thus, realizing this, I decided to swap that string with the element at either the position before the element which is larger than separator, or with the element that is larger than separator. However, after making these changes, I realized that the logic was unsound when the separator string is present multiple times within the array. At that point, I rewrote the separate function to order the strings in “ascending” order, which is an exponentially more elegant solution. Thus, it is simple to pinpoint what position in the array to return, and there is no conflict when the separator string is present in the array.

b.

1. int appendToAll(string a[], int n, string value);
   1. string people[5] = { "donald", "lindsey", "fiona", "rudy", "mick" };

int j = appendToAll(people, 5, "!!!");

* + 1. Simple test data to ensure appending is working properly for all n elements.

1. int lookup(const string a[], int n, string target);
   1. string officeholders[5] = { "donald", "lindsey", "mike", "adam", "nancy" };

int j = lookup(officeholders, 5, "adam");

* + 1. simple test data to ensure “adam” is found at the 3rd index
  1. string officeholders[5] = { "donald", "lindsey", "mike", "adaM", "nancy" };

int j = lookup(officeholders, 5, "adam");

* + 1. simple test data to ensure comparison between target and elements of array are case sensitive
  1. string officeholders[5] = { "donald", "lindsey", "mike", "adaM", "nancy" };

int j = lookup(officeholders, 0, "adam");

* + 1. test what happens when n is 0 – should return -1 because there are no elements to iterate through
  1. string officeholders[5] = { "adam", "lindsey", "mike", "adam", "nancy" };

int j = lookup(officeholders, 5, "adam");

* + 1. ensure smallest index of target is returned if there are multiple targets

1. int positionOfMax(const string a[], int n);
   1. string persons[6] = { "donald", "lindsey", "marie", "rudy", "fiona", "adam" };
   2. int k = positionOfMax(persons, 6);
      1. simple test data to return index of “rudy”
   3. string persons[6] = { "donald", "zindsey", "marie", "rudy", "fiona", "adam" };

int k = positionOfMax(persons, 6);

* + 1. changed max string element to ensure strings are compared correctly
  1. string persons[6] = { "rudy", "lindsey", "marie", "rudy", "fiona", "adam" };

int k = positionOfMax(persons, 6);

* + 1. two identical strings as the max string to ensure smallest index is returned

1. int rotateLeft(string a[], int n, int pos);
   1. string politician[5] = { "mike", "donald", "lindsey", "nancy", "adam" };

int m = rotateLeft(politician, 5, 1);

* + 1. simple test data to ensure item at index 1 is moved to the end, and all following elements are shifted left
  1. string politician[5] = { "mike", "donald", "lindsey", "nancy", "adam" };

int m = rotateLeft(politician, 5, 5);

* + 1. ensure when pos = n, pos is returned but nothing is changed in array
  1. string politician[5] = { "mike", "donald", "lindsey", "nancy", "adam" };

int m = rotateLeft(politician, 5, 0);

* + 1. same test as a, but with different value of pos
  1. string politician[5] = { "mike", "donald", "lindsey", "nancy", "adam" };

int m = rotateLeft(politician, 0, 3);

* + 1. to ensure that when n < pos, -1 is returned

1. int countRuns(const string a[], int n);
   1. string d[9] = { "rudy", "adam", "mike", "mike", "fiona", "fiona", "fiona", "mike", "mike" };

int p = countRuns(d, 9);

* + 1. array with 5 sequences, simple test to ensure basic functionality
  1. string d[9] = { "rudy", "adam", "mike", "mike", "fiona", "fiona", "fiona", "mike", "mike" };

int p = countRuns(d, 5);

* + 1. same test as above, but not all elements are iterated through in order to ensure this aspect is fine as well
  1. string d[9] = { "rudy", "", "", "mike", "fiona", "fiona", "fiona", "mike", "mike" };

int p = countRuns(d, 5);

* + 1. testing to make sure empty strings function correctly

1. int flip(string a[], int n);
   1. string folks[7] = { "adam", "", "fiona", "mike", "rudy", "nancy", "donald" };

int q = flip(folks, 5);

* + 1. flips first 5 elements, followed by last two
  1. string folks[7] = { "adam", "", "fiona", "mike", "rudy", "nancy", "donald" };

int q = flip(folks, 7);

* + 1. same as above, but all elements of array are flipped

1. int differ(const string a1[], int n1, const string a2[], int n2);
   1. string group[6] = { "adam", "", "fiona", "donald", "mike", "rudy" };

string folks[7] = { "adam", "", "fiona", "mike", "rudy", "nancy", "donald" };

int r = differ(folks, 7, group, 6);

* + 1. basic test to ensure iteration stops at pos 3

1. int subsequence(const string a1[], int n1, const string a2[], int n2);
   1. string names[10] = { "gordon", "marie", "nancy", "mick", "adam", "lindsey" };

string names1[10] = { "marie", "nancy", "mick" };

int t = subsequence(names, 6, names1, 3);

* + 1. simple test to ensure 1 is returned
  1. string names2[10] = { "gordon", "mick" };

int u = subsequence(names, 5, names2, 2);

* + 1. a2 is not a subsequence, -1 should be returned
  1. string names[10] = { "gordon", "marie", "nancy", "marie", "nancy", "lindsey" };

string names1[10] = { "marie", "nancy", "mick" };

int t = subsequence(names, 6, names1, 2);

* + 1. multiple subsequences to ensure lowest index is returned

1. int lookupAny(const string a1[], int n1, const string a2[], int n2);
   1. string names[10] = { "gordon", "marie", "nancy", "mick", "adam", "lindsey" };

string set1[10] = { "donald", "adam", "mick", "marie" };

int v = lookupAny(names, 6, set1, 4);

* + 1. basic test to ensure 1 is returned, as marie is in index 1 of names and present in set1
  1. string set2[10] = { "rudy", "fiona" };

string names[10] = { "gordon", "marie", "nancy", "mick", "adam", "lindsey" };

int w = lookupAny(names, 6, set2, 2);

* + 1. basic test to ensure -1 is returned when set2 elements are not present in names
  1. string names[10] = { "gordon", "marie", "nancy", "mick", "adam", "lindsey" };

string set1[10] = { "donald", "adam", "mick", "marie" };

int v = lookupAny(names, 2, set1, 4);

* + 1. test to ensure functionality even when n2 is larger than n1

1. int separate(string a[], int n, string separator);
   1. string persons[6] = { "donald", "lindsey", "marie", "rudy", "fiona", "adam" };

int x = separate(persons, 6, "gordon");

* + 1. basic test to split up a
  1. string persons[6] = { "donald", "lindsey", "marie", "rudy", "fiona", "adam" };

int x = separate(persons, 5, "gordon");

* + 1. same as above, but interesting elements is not the whole persons array
  1. string persons[6] = { "donald", "gordon", "marie", "rudy", "fiona", "adam" };

int x = separate(persons, 5, "gordon");

* + 1. ensures functionality when separator is present in persons
  1. string persons[6] = { "donald", "jordan", "marie", "rudy", "fiona", "gordon" };

int x = separate(persons, 5, "gordon");

* + 1. ensures functionality when separator is present in persons but is present in the portion of persons that is > separator